#19 PATENT

Attorney Docket No. 023071-553

1/2/98

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of:	
MICHAEL D. DOYLE et al.	Examiner: D. Dinh
Application No.: 08/324,443	Art Unit: 2317
Filed: 10/17/94	RESPONSE
For: EMBEDDED PROGRAM OBJECTS IN DISTRIBUTED HYPERMEDIA SYSTEMS	

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

The following is responsive to the Office Action mailed August 25, 1997, setting a response period expiring on November 25, 1997.

REMARKS

Claims 1,2, 5, and 44, 45, 48 are rejected over the Applicants' disclosed prior art (Mosaic + HTTP + HTML + "World Wide Web), referred to herein as "Mosaic," in view of Koppolu et al. The Examiner is thanked for extending the courtesy of an interview to one of the inventors, Professor Doyle, and his attorney, Mr. Krueger.

THE EXAMINER'S REASONING REJECTING CLAIM 1

The Examiner states that Mosaic does not have embed text format specifying an external object which automatically invokes an external application to execute and enable interactive processing within a portion of the browser controlled window.

Mosaic provides display and interaction with an external object by launching an associated program in a separate window.

It is further stated that Koppolu teaches a method for in-place interaction with a contained object and permits automatic display and interaction of a linked object in a compound document (i.e., hypermedia document) within a portion of a window controlled by a container application (i.e., the browser).

The Examiner concludes that it would have been obvious to apply the teaching of Koppolu to the processing of the hypermedia document of Mosaic because it would have provided a more integrated and expandable system for processing/viewing of linked objects in the hypermedia documents and reduces clustering of the window display. It is also concluded that since HTML is a text tag structure document encoding method, it is apparent that Mosaic as modified would have had a text tag for indicating links to an in-place interactive object.

THE INVENTION OF CLAIM 1

A browser application parses a hypermedia document to identify text formats in the document and responds to predetermined text formats to initiate processing specified by the text formats. The browser displays a portion of a first distributed hypermedia document in a browser-controlled window. The hypermedia document includes an embed text format, located at a first location in the hypermedia document, that specifies the location of at least a portion of an object external to the hypermedia document. The object has associated type information utilized by the browser to identify and locate an executable application external to the hypermedia document.

When an embed text format is parsed by the browser, the executable application is **automatically invoked** as a result of the parsing to execute on the client workstation.

When the automatically-invoked application executes, it displays the object and enables interactive processing of said object within a display window created within the portion of the hypermedia document being displayed.

SUMMARY OF THE CLAIM 1 ARGUMENT

The first part of the argument demonstrates that the cited art does not disclose or suggest several of the elements and limitations recited in claim 1. The second part of the argument demonstrates that the purpose, functions, and technology utilized in Mosaic and Koppolu are so different that, even if the missing features were disclosed in isolation, it would not have been obvious or even feasible for a person of skill in the art to combine the teachings of the reference to realize the claimed invention.

Turning to the first part of the argument, there is no disclosure or suggestion in Mosaic or Koppolu of automatically invoking an external application when an embed text format is parsed. Each of those references require user input, specifically clicking with a mouse pointer, to activate external applications to allow display and interaction with an external object.

As will be set forth in detail below, the object handlers in Koppolu (OLE) do not automatically invoke an external application to permit display and interaction with an object.

Turning to the second part of the argument, the functions and purposes of Mosaic and Koppolu (OLE) are fundamentally different.

On the one hand, Mosaic is directed to retrieving and displaying HTML documents received over the WWW, and to providing an efficient system for retrieving documents having links embedded in an HTML document being viewed. There is no provision within Mosaic for **editing** a document being viewed.

On the other hand, Koppolu (OLE) is directed to providing **editing** capabilities on a single user workstation for a compound document having embedded or linked objects which must be processed by applications external to a container object. When a compound document is opened, static bit maps of embedded or linked objects are displayed. An object to be edited must be selected by the user to invoke an external application to interactively process the object.

The different functions and purposes of Mosaic and Koppolu (OLE) are reflected in the different document structure. In Mosaic, since HTML documents are designed to be platform independent, the document structure is simple ASCII text. A browser parses a received document to identify HTML tags which specify various aspects of the document's appearance and links to other documents. In Koppolu, a container application creates a complex file structure which is utilized to render a document. There is no text parsing in Koppolu to render the compound document.

The complex binary data file structure of an OLE compound document file can only be created and utilized by platform-dependent OLE enabled applications. In contrast, an HTML document can be created utilizing the simplest ASCII text editor and can be viewed by any browser on any platform connected to a network. Furthermore, OLE is designed for manipulating data which is entirely resident on the user's machine, while Mosaic is designed for viewing data files which are typically remotely-networked. OLE teaches no facility for working with remotely-networked components, and, in fact, employs an architecture which would most likely have hindered Mosaic's capabilities in this regard. (For a detailed description of how Koppolu works see below, Argument at II.C.2.)

Thus, a person of skill in the art, considering the teachings of Mosaic and Koppolu (OLE) at the time of the invention, would not have conceived of the present invention. To so combine the references would require ignoring the purposes, functions, and document structures of both references.

As stated by Judge Markey:

> "The third fundamental error was the refusal... to credit the real world environment surrounding the inventions disclosed in the reference patent and in the patents in suit.
>
> It must be remembered that the Examiner is required to consider the references in their entireties, i.e. including those portions that would argue against obviousness ..."

> The Court specifically stated "I don't care how things work or don't work." But "how things work" is critical to encouragement of every research and development activity, and every advancement of the arts useful to the public, the very purposes of the patent system.
>
> It was a refusal to consider "how things work" that caused the court to cite isolated minutia from various references, while ignoring critically important structural **distinctions** that significantly affect the different achievements of which the references and claimed structures are capable. Panduit Corp. v. Dennison Manufacturing Company, 227 USPQ 337, 345 (CAFC 1985).

Turning to the real world environment, as discussed at the interview, Microsoft Corporation had the rights and technology to both Mosaic and OLE but did not conceive of the claimed combination or attempt to combine the features of Mosaic and OLE. Instead, Microsoft developed ActiveX, which does not employ a combination of OLE and the teachings of Mosaic, but utilizes the claimed features of the invention, including an embed tag text format as defined, which is parsed to cause the automatic execution of an external application to display and interactively process an external object within the browser controlled window. These claimed features have been adopted by the entire industry and lauded as a major breakthrough in Internet and computing technology.

All the above factors compel a conclusion that the claimed combination is patentable. As described above, the browser application is not analogous to an OLE container application and an HTML document is not analogous to an OLE compound document. Equating a browser and an HTML document to container application and an OLE compound document requires impermissably selecting isolated features from Mosaic and OLE, utilizing Applicants' disclosure as a roadmap, while ignoring important structural and functional differences relating to the purposes and implementations of the systems disclosed.

DETAILED ARGUMENT

TABLE OF CONTENTS

CLAIM 1 ARGUMENT

PART I. The proposed combination does not show the features of the Applicants' invention. There is no suggestion or teaching in Koppolu (OLE) of modifying Mosaic to automatically invoke an external application, when an embed text format is parsed, to display and interactively control an object in a display window in a document being displayed in a browser controlled window.

A. Mosaic

Mosaic parses a received document, passively displays links from text or picture elements of a first hypermedia document to other external data objects, and retrieves information identified by a link when a user interactively selects the link. The retrieved information either replaces the first hypermedia document, or is displayed in a separate window other than the window displaying the hypermedia document. Mosaic has the capability of allowing the user to interactively invoke an external application to open a new window to display file types that cannot be displayed by Mosaic (helper applications).

Mosaic launches helper applications in response to a user's interactive command, in a separate window to view certain types of file types. As described in the specification, the mechanism for specifying and locating a linked object is an HTML anchor "element" that includes an object address in the format of Uniform Resource Locator (URL). (Application at pg. 3, line 30).

Many viewers exist that handle various file formats such as ".TIF," ".GIF," etc. When a user commands the browser program to invoke a viewer program, typically by clicking on an

anchor with a mouse, the viewer is launched as a separate process. The viewer program displays the full image in a separate "window" (in a windowing environment) or on a separate screen. This means that the browser program is no longer active while the viewer program is active. The viewer program is completely independent of the browser after being invoked by the browser so that there is no communication between the viewer program and the browser program after the viewer program has been launched.

As a result, the viewer program continues to run, even after the browser program execution is stopped, unless the user explicitly stops the viewer program's execution.

Mosaic was a significant advance that made the WWW easily accessible and gave document authors a powerful tool to provide simplified user-activated access to viewing of hypermedia documents and related external data objects anywhere on the WWW network.

B. Koppolu (OLE)

1. General

Koppolu's OLE system provides a method for interacting with a contained object within the window environment of a container application of a container document. When a user interactively selects a bitmapped image of the contained object, the method integrates a plurality of the server resources with the displayed container window environment. When the user then interactively activates the previously-selected object image, OLE invokes a server application to process the original data referenced by the contained object image. Since OLE was designed for integration of very large programs, a facility is provided whereby the server application can conserve space on the computer display by integrating the server application's menu and GUI system with that of the container application.

The designers of OLE did not attempt to include the capability to have the entire server application (data display, menus and GUI) run within the embedded window, (as is enabled by the Applicants' invention), since such a system would have been

impractical with large monolithic applications, and to incorporate such a feature would have required a total redesign of the object activation and event handling system of OLE.

Furthermore, OLE does not **parse** text tags in the document in order to render the document (as required by the Applicants' claims). OLE uses an internal binary pointer mechanism to provide for accessing the referenced source data structures that contain the document objects. The actual linking mechanism between the container document and the containee server application is coordinated by the operating system's registry database. Therefore, the document text is not used to determine contained the object's type. At the time of initial object selection by the user, and prior to server application launching, OLE references the operating system's global registry database in order to identify which server application is related to a particular data object and to determine what interactive operations are provided by the relevant server application.

Koppolu et al. describes a method referred to as in-place interaction, for integrating the functionality of two application programs within the context of a compound document. It allows the user to interact with OLE-embedded or OLE-linked data within the context of the compound document. This method is embodied in the Object Linking and Embedding (OLE) API from Microsoft Corporation.

Koppolu defines a **compound document** as "a document that contains information in various formats. For example, a compound document may contain data in text format, chart format, numerical format, etc." Embedded data is then defined as follows: "Data that is copied from the clipboard into a compound document is referred to as '**embedded**' data. The word processing program treats the embedded data as simple bitmaps that it displays with a BitBit operation when rendering the compound document on an output device." Koppolu goes on to define "**linked**" data by stating: "Some prior systems store links to the data to be included in the compound document rather than actually embedding the data. When a word processing system pastes the data from a clipboard into a compound document, a link is stored in the

compound document. The link points to the data (typically residing in a file) to be included. These prior systems typically provide links to data in a format that the word processing system recognizes or treats as a presentation format (such as a bitmapped image of the rendered data)."

When a document containing embedded or linked data is loaded into the application which displays the compound document, a bitmapped **picture** or a vector **drawing** (metafile) of the data (the "**object presentation format**," Col. 33, line 60) is typically displayed at a location in the document, and the user can indicate selection of the object to the word processor application by, for example, interactively single-clicking on the picture of the budget data which is shown in the document. An example is given of a spreadsheet object, containing budgeting data, that is embedded in a word processor document.

The embedded or linked data is made accessible to the user through a process called "activation in-place." As Koppolu describes, "when embedded or linked (contained) data is activated in place, the menus of the application that implements the contained data are merged with the menus of the application that implements the compound document to create a composite menu bar." Koppolu states, "when the user decides to edit the budgeting data, **the user selects the spreadsheet object** and requests the word processing application to edit the object (e.g., by double clicking on the object using the mouse). **The word processing application then launches the spreadsheet application**, requesting that it edit the spreadsheet object. The spreadsheet application negotiates with the word processing application to edit the spreadsheet object, using windows and the menu bar of the word processing application. This process of launching the server application is called "activation." In this example, the word processor is described as a "container application" and the spreadsheet application is termed a "server application." It is important to note here that the container document does not automatically launch the server application at document-rendering time, but rather, the **user must issue two separate interactive**

commands after the time of document rendering in order to cause the container application to invoke the server application.

OLE provides a limited and specific set of methods to allow user interaction with embedded or linked objects within compound documents. As stated in "Inside OLE 2.0" (hereafter referred to as "Brockschmidt") by Kraig Brockschmidt, published by Microsoft Press in 1993 (ISBN 1-55615-618-9):

> "Creating new objects and displaying or printing them doesn't do anything more for us than a static bitmap or metafile. One of the most important features of compound documents under OLE is that you can activate the object, that is, ask it to execute one of its verbs and thus perform some action such as playing a sound or opening its data for editing. This ability for activation is the only thing that separates an embedded or linked object from a static one. But some way or another you have to allow the end-user to select the actions available for that object which varies object to object. For this reason the OLE 2.0 user interface defines two methods to allow end-users to invoke verbs.
>
> The first method is to execute what is known as the primary or default verb when the object is double-clicked with the left mouse button. The exact meaning of this primary verb is defined by the object, not by the container, so the container blindly tells the object to execute without any knowledge of what will happen. The second method, of which the container is equally ignorant, is to provide a menu item on the container's edit menu that lists all the available verbs on the currently selected object. When the end-user selects one of these menu items, the container again blindly tells the object to execute a verb. However, the container can ask the object to perform known actions by using predefined verbs, although these options are generally not shown directly to the end user."

Brockschmidt sums up the three possible states of a containee object in the Summary section of Chapter 9 by stating:

> "An object in a container document may have three states: passive, loaded, and running. A passive object exists entirely on disk and is not visible, printable, or available for any manipulation. When an end-user opens the document in which the object lives and the container application loads the object, it transitions to the loaded state where it may be seen and printed but not edited or otherwise manipulated in any way. **Only when the object is activated does it transition to the running state where the user may perform any number**

of actions on that object, such as playing or editing the data."

Thus, OLE was an advance that allowed a user to create and edit a single compound document while calling upon the capabilities of a plurality of large, monolithic stand-alone applications and having those applications share a single container graphical user interface environment, thereby conserving display space and allowing the user to focus more directly on the document editing process.

2. OLE Object Handlers

During Applicant's interview with the Examiner on November 6, 1997, the Examiner asserted that since OLE shows an object handler which is dynamic-link library (DLL) code that can automatically be invoked at document rendering time, and since this object handler DLL code may be considered to be part of the server application, then it would have been obvious to enhance Mosaic by providing such a DLL object handler that would be automatically invoked at document rendering time to provide display and interactive processing of the object within a window in the hypermedia document. Applicants respectfully assert that such reasoning is incorrect.

As Koppolu states, "the invoking of a server application can be relatively slow when the server application executes as a separate process from the container application. In certain situations, this slowness may be particularly undesirable, such as, for example, if the user wants to print a compound document that includes many containee objects." Such a problem is solved in the Koppolu system by providing special code that is loaded when the word processor application is launched, where that code provides a subset of the functionality of the full server application. An example would be code which allows printing of embedded or linked spreadsheet data, without having to start up the spreadsheet application itself. This code is called an **"object handler."**

Koppolu does not give much detail about the functioning of object handlers, which are shown to be implemented as dynamic

link libraries (DLLs). Full details on object handler implementation and functionality can be found in Brockschmidt.

Brockschmidt comments on object handlers (Chapter 11, section: "Why Use a Handler?"): "There are two main reasons for implementing an object handler to work with your local server: speed and portability. First, an object handler can generally satisfy most requests a container might make on an object such as drawing an object on a specific device or making a copy of the object in another IStorage. Object handlers may also be capable of reloading a linked file and providing an updated presentation to the container. **Object handlers do not, however, provide any sort of editing facilities** for the object itself."

If the entire server application is implemented as an in-process DLL, this is called an "in-process server." The above statement by Brockschmidt shows that the use of the term "object handler" relates specifically to a limited set of object-related facilities that can be automatically invoked by the container application at document rendering time, but which **do not include** capabilities for interactively processing object data. Interactive processing of object data can only be accomplished through interactive invoking of either an in-process server or a local server (standalone executable). In-process servers do allow editing capabilities for object native data, but these editing capabilities are invoked **only after the containee object has been interactively activated by the user,** as described below. This is further supported by the teaching of Koppolu that user interaction with containee objects is provided by OLE only after interactive activation of the containee object server by the user (Col. 7, lines 56-66).

Brockschmidt goes on to describe the problems associated with use of both object handlers and in-process servers. These include 1) limited interoperability, even across different versions of OLE and different versions of Intel processors, and 2) the lack of message loops in DLLs, drastically limiting use of interactive capabilities such as keyboard accelerators. According to Brockschmidt, "The other technical issue of an in-process server specifically (but not a handler) is that since

there is nothing that can ever run stand-alone (like a local server EXE can) **there is no possibility to provide linked objects.**" Brockschmidt explains this by pointing out that in-process DLLs cannot access files external to the compound document file. This limits the use of in-process servers to working with embedded (encapsulated) data stored ***within*** the container document file, rather than linked ***external*** data files.

To repeat, Brockschmidt clearly states that "an object handler can generally satisfy most requests a container might make on an object such as drawing an object on a specific device or making a copy of the object in another IStorage. Object handlers may also be capable of reloading a linked file and providing an updated presentation to the container. **Object handlers do not, however, provide any sort of editing facilities** for the object itself." Brockschmidt goes on to emphasize: "When an end-user opens the document in which the object lives and the container application loads the object, it transitions to the loaded state where it may be seen and printed but not edited or otherwise manipulated in any way. **Only when the object is activated does it transition to the running state where the user may perform any number of actions on that object, such as playing or editing the data.**"

C. Conclusion

The Examiner has stated that Mosaic does not have an embed text format specifying an external object which automatically invokes an external application to execute and enable interactive processing with a portion of the browser controlled window.

In view of the above, it is clear that the Koppolu (OLE) reference does not disclose or suggest the missing features. In OLE, when a compound document is opened, static pictures of included objects are rendered in presentation format. Invoking of an external application requires a user-activated selection of the object. The object handlers provide no interactive control of a displayed object.

PART II. The selection and combination of features of the claimed invention from references such as Mosaic and Koppolu is only possible by disregarding the striking differences in the structure and operation of the references. Such disregard is only possible by utilizing the applicant's disclosure as a roadmap to randomly select and combine features from the references.

A. Introduction

Mosaic was designed as a **network communications and information retrieval** tool for **viewing** static documents. OLE was designed to integrate the **document creation and document editing** capabilities of multiple **non-networked** monolithic applications whose primary purposes were to act as stand-alone applications, and to accomplish such integration primarily through extensive modification of the user interface of a single container application. OLE is based upon a **document-centric** paradigm, where the document creation and editing applications move around a single centered document, rather than moving the document around to each application separately. The applications shown in these two references were designed for **fundamentally different** purposes, and neither was designed to solve the problem that the claimed invention solves. There is no suggestion or motivation in either Mosaic or Koppolu's OLE system that would lead one to attempt a combination of the two to achieve the features of the claimed invention.

B. The Applicable Law

1. Impermissible Hindsight

The test for obviousness is whether the subject matter of the claimed invention would have been obvious to one skilled in the art at the time the invention was made, not what would be obvious to an Examiner after reading the Applicants' disclosure.

Panduit Corp. v. Dennison Manufacturing Company, 227 USPQ 337, 343 (CAFC 1985).

It is impermissible to ascertain factually what the applicant's did and then view the prior art in such a manner as to select from the art random facts only those which may be modified and then utilized to reconstruct the applicant's invention from the prior art. Panduit Corp. v. Dennison Manufacturing Company, 227 USPQ 337, 345 (CAFC 1985).

2. Combining References

A suggestion for combining the references must appear in the prior art. It is required to consider the references in their entireties, i.e., including those portions that would argue against obviousness. Panduit Corp. v. Dennison Manufacturing Company, 227 USPQ 337, 345 (CAFC 1985).

Additionally, the real world environment surrounding the references must be considered. The Examiner must consider "how things work" when considering whether the references would have made the claimed combination obvious at the time of the invention. It is necessary to consider what workers of ordinary skill in the art actually did. Panduit Corp. v. Dennison Manufacturing Company, 227 USPQ 337, 345 (CAFC 1985).

C. How the Systems Described in the References Work.

1. How Mosaic Works.

The World Wide Web was designed as an information retrieval and hypertext document viewing system. A major priority in the design of the Web was that the documents would be viewable and similarly navigable irrespective of the type of computer on which the document was viewed. That is, the **look and feel** of the document would be similar on as many different types of computing platforms as possible. The document creation and dissemination model was **write once, publish many.** This refers to the fact that a Web document author would create a single document file and publish that file merely by making it accessible on an Internet server. An unlimited number of users could then retrieve and view that document by simply entering the

Internet address of the document (the URL) into their Internet-connected Web browsers. The browsers would use a simple request/response protocol to retrieve specified documents and related data from remotely-networked Web server programs.

In order to insure cross-platform uniformity of document appearance, the document was defined through the use of ASCII text, where specific text formats, otherwise known as "tags," would be used within the document text to specify various aspects of the document's appearance and linkages to other documents or related data. Each browser, therefore, incorporated a parser which would distinguish the formatting tags from the document's narrative text, classify those tags into pre-defined categories, break each tag into its basic components, and then invoke appropriate browser subroutines to respond appropriately to the meanings of the tag components. Although the browser subroutines were built from machine-specific native code, this text tag mechanism allowed the design of a variety of browsers for various computing platforms that could respond in similar ways to similar types of text tags, and therefore result in similar-appearing documents on dissimilar computers. A binary document data format was avoided in order to promote cross-platform compatibility, due to the variation in binary data handling methodologies on various different operating systems, and to simplify the requirements for document creation tools. All that a Web document author needs in order to create a Web document file is a simple ASCII text editor, which is a pre-existing application in all commonly-found operating system packages.

Because of the write-once-publish-many paradigm of the Web, all users would see the same basic document content that the document author originally designed. End users, the remotely-networked individuals using Web browsers, were allowed to view the document, but could not edit the source document in any way. To allow user-editing of the original document text would have been counter productive, since it would be contrary to the write-once-publish-many design philosophy of the fundamental Web architecture.

Mosaic was an advance that provided a graphical environment within which to view and navigate Web documents. Mosaic allowed Web documents to display text in a variety of fonts and type styles, as well as to display static bitmapped images on the page with the text. Hypertext links to external documents could be displayed by rendering linked text in a distinguishing color, or by outlining a linked image with the same distinguishing color (usually blue).

Unusual data formats that were not supported by the native rendering subroutines of the browser could nevertheless be made available to the user by encoding a link in a Web document directly to the data file in question. The type of the data was designated through the use of a data-type-specific filename extension, such as .TIF or .MPG. If the user had pre-installed a viewer application that was capable of viewing that unusual data type, then Mosaic could be configured to allow the user to specify downloading of the data and launching of the external viewer application by clicking on a link to the data object. This would allow viewing of the data in a window external to the Mosaic application window.

This link-based data retrieval for viewing documents and related data objects was designed specifically for viewing document-related data, not for end-user editing of that data. This applied to the external viewer applications as well. Although one could, in theory, configure a data editing program as an external viewer, to actually edit the downloaded data would not have made much sense, since those edits would have only affected the local temporary copy of the document data, and would have had no effect on the original document files on the remote Web server. This design principle is reflected in the file caching architecture of Mosaic. When document text, image data, or external object data are downloaded by the browser, these data are stored in local temporary cache files, to speed up document rendering for later requests for the same data. Since these data are only intended to be viewed, and not edited by the end-user, these cache files are only temporary, and are routinely automatically purged by the Web browser after a certain amount of

time, or after it is determined that the user is not likely to need them again. This makes more efficient use of the end-user's disk drive space over time, preventing cluttering up of the disk drive with unneeded files. To provide editing capabilities for these files would have been nonsensical, since end-user edits would automatically be lost at the subsequent cache purge.

Mosaic was therefore not designed to allow interactive manipulation of object data. On the contrary, it appears to have been specifically designed to ***discourage*** such object data manipulation. External viewer programs could, theoretically, allow such manipulation of object data, but the fundamental design of Mosaic shows that such manipulation was intended only for viewing of temporary local copies of object-related data.

2. How Koppolu (OLE) Works

The OLE system, as defined in Koppolu and as embodied in OLE 2.0 from Microsoft Corporation, was designed primarily as an environment for compound-document creation and editing. Through OLE-based applications, users create compound documents by transferring object data through the clipboard during the document creation process. That object data is then used to create either an embedded object, where the contained object data is encapsulated in the container document file, or linked data, where the object data is stored in a separate file. In both cases, the container document file stores a presentation format, or picture, of the object data. OLE Compound Files store the necessary information in the container document file for rendering the various document components and store object data for embedded objects. The file structure was designed as a simulated file system in order to facilitate incremental fast saves and incremental viewing of large documents during the document creation and editing process. These data are stored in platform-specific binary format to accelerate the task of reading and writing necessary data. Since the document is represented in an hierarchical ordered data structure, and since object-specific object handlers determine how various document components are

rendered, there is no need to parse the document in order to accomplish rendering.

The Microsoft Dictionary of Computer Terms defines the term "parsing" as follows: "To break input into smaller chunks so that a program can act upon the information. Compilers have parsers for translating the commands and structures entered by a programmer into machine language. A natural-language parser accepts text in a human language, such as English, attempts to determine its sequence structure, and translates its terms into a form the program can use. Data base management programs and expert systems often support natural-language parsing. A user could ask such a program to display the relationship between inflation and home buying in the last decade." The program might break the sentence apart and interpret it in the following way:

display

Present the results as a chart.

the relationship between

Do a linear regression analysis.

inflation and home buying

The independent and dependent variables, respectively.

in the last decade

Use data from 1980-1989.

Comparing OLE binary data formats to Mosaic's ASCII text tag mechanism, from the point of view of parsing, would be similar to comparing machine-code programming to the use of a higher level programming language. Similar to OLE's document data files, machine code programs are stored in a binary data structure format that is specifically tailored to the computer processor architecture. They are especially efficient, since they do not require a parser for execution. Use of Mosaic's tag-parsing mechanism, however, is more like using a higher-level programming language. What is given up by Mosaic in terms of run-time performance is recouped through ease of document

development, simplification of browser design, and cross-platform compatibility for document viewing.

OLE uses two binary data structures to store objects in compound documents' IStorage and IStream. The IStorage data elements correspond to analogs for directories, and the IStream data elements are file analogs. These data structures store the document objects, as IStreams, in the order in which they appear in the document. Each document element, such as a paragraph or an embedded object, has an associated object handler. When the document is rendered by the container application, this rendering is merely a matter of invoking the objects in the order in which they appear. There is no need to **parse** the document file, to break it into components and to classify those components (as in Mosaic), since the objects are, by definition, already associated with the methods necessary to render them.

Since cross-platform compatibility is not a priority in the OLE system, the fundamental architecture of OLE argues ***against*** the idea of modifying OLE to allow such parsing for compound document rendering. This is because parsing the document would slow down the performance of the OLE system, and because of the fact that parsing in OLE would provide redundant and unneeded capabilities, since, by definition, each document object element in OLE already has an object handler associated with it to handle the rendering chores.

Another consequence of the OLE architecture is that document rendering of containee object data is closely bound to the specific application methods that were used for creating that object data. In order to manipulate OLE embedded or linked object data, the user must have a copy of the application that originally created that data. This is appropriate for a system that was designed specifically to ease the document creation and editing tasks for a single user working on a single computer, but it would drastically reduce the usefulness of a program such as Mosaic that is intended for the viewing of documents that are simultaneously being widely distributed over networks to heterogeneous populations of other client workstations.

OLE simplifies the task of the compound document author, by allowing interactive editing of containee object data from within the same windowing and graphical user interface (GUI) environment of the container application. In this way, the document author can invoke large external object-editing applications which can take over the GUI of the container application, rather than forcing him or her to pop up another set of application windows in order to edit the document. Since OLE was designed to integrate a group of already-popular large programs, such as Microsoft Excel and Microsoft Word, OLE provided the advance that allowed these applications to be interactively invoked to change the GUI environment that surrounded the compound document so that the user would not have to move his or her attention to an external windowing environment in order to edit the object. Since these external applications were intended to be invoked only for editing purposes, and since that invocation inevitably resulted in a modification of the container application's GUI, it made sense that OLE containee server applications could ***not*** be automatically invoked to allow interactive processing of object data. In fact OLE forced the user to make not one but ***two*** interactive commands prior to server invocation, thereby reducing the possibility that one of these large external applications would be inadvertently invoked.

In order to facilitate the ability of several large and complex applications to share each other's GUIs, OLE requires that all interaction with containee server application methods occur by way of the container application's internal message handling subroutines. When a given containee object image (a bitmap or metafile of linked or embedded data) is interactively selected by the user within the container document, OLE then invokes functions in order to allow the container application to determine how to modify its graphical user interface and menu structure for a particular type of embedded data. This occurs prior to invoking the server application. OLE determines from the operating system's global registry what actions are supported by the server application implementing the selected object, and then displays the actions in a menu. The user must then issue

yet another interactive command before the OLE object becomes activated and available for interactive processing. Koppolu states: "Once a user has selected a desired action (from the menu or by double clicking on the object), the container application can then invoke the server application by passing it an indication of the action to perform on behalf of the container application."

After in-place activation, the user then interacts with the server application by indicating to the container application the actions that are desired to be performed on the contained data. Koppolu states: "Once the user has activated an object in place, the user interacts with the object within the container application by selecting actions through the menu bar of the container application (which is the composite menu bar). Because some of the menus belong to the server application and others belong to the container application, the window procedure for the container application frame window must decide whether to send the menu input event to a function within the container application or within the server application. **Thus, all messages received by the container application that correspond to its frame window are thereafter routed fast to the special message handler. This special message handler then decides to which application to route the message event received.**"

Koppolu teaches that OLE-based applications provide users the capability to interact with containee objects via the menu and messaging system of the container application. OLE does not teach that the user interacts directly with an embedded application's user interface within a window ***in*** the document, as in the Applicants' invention. Koppolu's OLE system does not teach interactive processing of the contained object ***within*** the embedded object's window, as required by the claims of the Applicants' invention, since OLE teaches that the user should employ the modified menu system, and the messaging system, of the container application in order to process the contained object's data.

D. The way Both Mosaic and Koppolu work therefore teach away from a combination of the two

Mosaic teaches that document tag parsing, rather than binary data representations, should be used to specify the elements and layout of the hypermedia document. Mosaic further teaches that the content of a hypermedia document should not interfere with the functionality of the hypermedia browser application. For example, an important feature of Mosaic was the ability to interrupt the downloading of a partially-rendered long document. OLE would interfere with this, feature since an activated server would deactivate the GUI (including the Stop and Back buttons) in Mosaic.

As described in detail above, Koppolu teaches away from the method of parsing the text of a hypermedia document in order to identify text formats which specify the locations of contained object data. OLE teaches that the various component data elements of a document should be stored as binary data structures that contain objects that are invoked in storage order, to render their presentation formats on the screen or printed page in a binary representation. In fact, OLE employs an emulated file system, within the OLE compound document file format, to organize and store component data. This scheme facilitates fast incremental saves of document component data during the OLE document editing process, but would be totally unworkable in Mosaic.

Koppolu also teaches away from the use of OLE for networked data distribution. There is no provision, suggestion, or motivation in Koppolu to provide for automatic invocation of a server application to allow interactive processing of object data when a container document is viewed. Furthermore, there is no suggestion in either Mosaic or Koppolu of modifying Mosaic so that an external application, by analogy to Koppolu the server application, is automatically invoked at the time of Web document rendering to display and enable interactive processing of the object within an embedded window within the Web document.

The Applicants' invention allows the networked hypermedia document to act as a coordinator and deployment mechanism, as

well as a container, for any arbitrary number of external interactive data/application objects, irrespective of where those objects are located on a network, while hiding the details of such coordination and deployment from the documents reader (user) as the reader uses and interacts with the various data/application objects on a variety of computer platforms. This allows the networked hypermedia document to act as a platform for entirely new kinds of applications that could not have been possible before the invention.

Because most of the functionality exposed to the user is defined most directly by the hypermedia document itself, rather than any specific computer operating system or document container application, document-based applications using the claimed invention tend to have the same look and feel to the reader, regardless of what type of computer or operating system is being used to run the operating system, and regardless of what type of browser is being used to view the document.

Additionally, because the claimed embed text formats in the document cause the browser to automatically invoke the external application, the hypermedia ***document*** itself, and by implication the author of that document, ***directly*** control the extension of the functionality of the browser. As a consequence of the features of the claimed invention, the ***document***, rather than the browser, **becomes the application;** that is, the document together with its embedded program objects, exposes all the functionality that the user needs to interact with and process the entire content of the compound hypermedia document.

E. What Workers of Skill in the Art, Aware of and in Possession of Rights to Mosaic and Koppolu (OLE) Actually Did

1. The owner of the commercial rights to both the technology cited in Koppolu and the applicant-cited prior art (Mosaic) chose to employ features of the claimed invention, rather than to attempt the combination proposed by the Examiner

During the 2 years after the date of the Applicants' patent application, Microsoft Corp. obtained the commercial rights to the Mosaic Web browser and, of course, developed many applications based upon the OLE technology disclosed in Koppolu. The software company then set out to implement improvements to Mosaic to allow the use of embedded interactive applications within Web pages. However, Microsoft did not combine the features of Mosaic and OLE but instead developed ActiveX. The fact that ActiveX is not a combination of Mosaic and OLE is made clear by Microsoft itself in a conversation between Web Technique's editor and the ActiveX product managers from Microsoft:

> --So tell me about ActiveX. Isn't is just OLE for the Internet?
> --Wrong,... ActiveX is a new API that, like OLE is based on Microsoft's Component Object Model (COM). While OLE supports a compound document structure for desktops, ActiveX is designed specifically to embed rich media objects within Web-based documents.
> From "ActiveX, a Standard? by Michael Floyd (Editor in Chief), WebTechniques, October 1996, page 5.

As described below, ActiveX utilizes an embed text format, adopted from the present invention, and employs all the features of applicant's claim 1 to function as a universal wrapper to bridge the gap between a browser and OLE applications as well as Java, VRML, Shockwave, Visual Basic, C++, and other scripting tools.

The Microsoft Press book, ActiveX Controls Inside Out, by Adam Denning, 1997 (ISBN 1-57231-350-1), shows specifically that ActiveX employs the features of the claimed invention for the implementation of embedded program objects in distributed hypermedia documents, rather than to attempt to use the combination proposed by the Examiner.

Specifically, the Denning reference shows, in Chapter 13, page 402, that the ActiveX system uses an embed text format (<OBJECT>), at a location within the hypermedia document, that specifies the location of at least a part of an object external to the hypermedia document (using the CODEBASE parameter), wherein the object has type information associated with it (the CLASSID) that is utilized by the browser to identify and locate an executable application external to the hypermedia document, and wherein the embed text format is parsed by the browser to automatically invoke the external application to execute on the client workstation in order to display the object and enable interactive processing of the object within a display window (defined by the WIDTH and HEIGHT parameters) created at the embed text format's location within the distributed hypermedia document being displayed in the browser-controlled window, as required by Claim 1. That the ActiveX embedded program object **internally** supports a modified OLE-type API for communication between the ActiveX control and other OLE-compatible applications is **irrelevant** to the patentability of the claimed invention.

The fact that Microsoft chose to implement a system (ActiveX) that employs the features of the Applicants' invention rather than to perform a combination of OLE and Mosaic as proposed by the Examiner, even though Microsoft would have had a strong incentive to make such a combination if it were obvious, feasible and useful, provides **direct and undeniable evidence that such a combination was not obvious and would not have improved Mosaic**.

F. How the Real World Responded to Applicant's Invention

1. The commercial success of products developed subsequent to filing the present application, incorporating the claimed features, established that the combination was not obvious at the time of the invention.

As is shown in the Inventor's declaration filed 6-2-97, several major competitors have incorporated the features of

Applicants' invention, rather than to use the techniques of the prior art. The most notable of these products include the ActiveX applet system from Microsoft corporation, the Navigator Web browser application from Netscape corporation and the Java Web applet system from Sun Microsystems corporation. The Inventor's declaration further shows that the success of these products is directly attributable to the features of the claimed invention which each of these products incorporate, including an embed text format that is parsed by a Web browser to automatically invoke an external executable application to execute on the client workstation in order to display an external object and enable interactive processing of that object within a display window created at the embed text format's location within the hypermedia document being displayed in the browser-controlled window. The developers of these products chose **not** to attempt the proposed combination, **even though the OLE technology was widely available for development purposes at the time of the Applicants' invention**. Rather, they chose to implement the features of the Applicants' invention, in a manner that is **virtually identical** to the preferred embodiment disclosed in the Applicants' specification, in making improvements to Mosaic.

Some of these competitors have made laudatory statements about the elements of the Applicants' invention which are incorporated into their respective products, and have characterized those features as being a significant and innovative advance over prior art techniques.

It is well known that, in the 1966 case of Graham v. John Deere, the U.S. Supreme Court decreed that Section 103 is to be interpreted by taking into consideration secondary and objective factors such as commercial success, long-felt but unsolved need, and failure of others.

As is shown in the 6/2/97 Inventor's declaration, there is universal acceptance within the computer software industry that the aforementioned products incorporating the features of the claimed invention have attained an extremely high degree of commercial success. Java, Navigator and ActiveX (all products incorporating the features of the claimed invention)

represent among the most popular current technology in the computer industry for new application development, and are among the most successful products in the history of computer software.

2. The products by others incorporating the features of the claimed invention have been given many awards and have received considerable recognition in professional publications.

As is shown below and in the 6/2/97 Inventor's declaration, Netscape, ActiveX and Java, all incorporating features of the Applicants' invention, have each been lauded as among the most innovative (and therefore non-obvious) technologies to appear in the computer software industry in recent years.

Some examples are:

The 1996 Discover Awards for Technical Innovation -- to Java and Navigator

PC Magazine 1995 Technology of the Year Awards -- to Java and Navigator

New Media Magazine 1996 Hyper Awards -- to Java and Navigator

New Media Magazine 1997 Hyper Awards -- to Microsoft's ActiveX applet system

As is evidenced in the 6/2/97 Inventor's declaration, this acclaim is due to the innovative nature of the features of the claimed invention incorporated into those products, argues strongly against the obviousness of the pending claims.

DEPENDENT CLAIMS

Claims 2-5 which depend on claim 1 are allowable for the same reasons as set forth above and are further allowable because those claims recite additional limitations not disclosed in the cited references.

In particular, claim 3 recites the additional steps over Claim 1 of "interactively controlling said controllable application by interprocess communications between the browser and the controllable application" (claim 2); "continuing to exchange communications after the controllable application has

been launched" (claim 5), and "additional instructions for controlling said controllable application reside on said network server, wherein said step of interactively controlling said controllable application includes the following sub-steps: issuing, from said client workstation, one or more commands to the network server; executing, on said network server, one or more instructions in response to said commands; sending information from said network server to said client workstation in response to said executed instructions; and processing said information at the client workstation to interactively control said controllable application."

None of the cited references show these features. As described above, in Mosaic there is no disclosure of interacting with an external controllable application. Further, in Koppolu (OLE) the client and server applications are on the same computer. There is no provision for interacting with an application on a network server.

The reference Moran discloses a tool for interactive visualization of large, rectilinear volumetric data called Tele-Nicer-Slice-Dicer (TNSD). TNSD is based on client-server design where the client-side process is an extended version of a stand-alone visualization tool and the server process runs on a high-performance system where the data are located.

The client-side process describes data sets by text fields. Each data set description is used as a command which is sent to the server when a volume from the corresponding data set is requested. The use of a remote server is transparent.

The only relevance of Moran to the subject matter of claim 3 is the use of a network. There is no disclosure relating to HTML, the WWW, or embedding controllable objects in a document displayed in a browser-controlled window.

The teachings of Mosaic, Koppolu (OLE), and Moran would not make the combination of claim 3 obvious. Such a combination is in violation of the requirement of 35 U.S.C. §103 because the combination requires taking isolated features from the references, utilizing applicant's disclosure as a roadmap, while ignoring the operation and purposes of the references.

Claim 4 recites the additional steps over Claim 3 "wherein said additional instructions for controlling said controllable application reside on said client workstation."

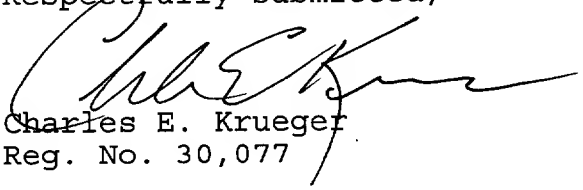
None of the claimed references show this feature. This feature produces the additional surprising and unexpected results of enabling a client and network server system to be self-contained on the client workstation.

Claims 44-48 are apparatus claims of the same scope as claims 1-5 and are thus allowable for the reasons recited above.

In view of the foregoing, Applicants believe all claims now pending in this application are in condition for allowance. The issuance of a formal Notice of Allowance at an early date is respectfully requested.

If the Examiner believes a telephone conference would expedite prosecution of this application, please telephone the undersigned at (415) 576-0200.

Respectfully submitted,



Charles E. Krueger
Reg. No. 30,077

TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, 8th Floor
San Francisco, California 94111-3834
(415) 576-0200
Fax (415) 576-0300
CEK:db
i:\cek\share\02307i\553\dec23.res